Parallel Programming Project

n-body: Move U'r Body

Tara Aggoun
21304625

Riwan Coëffic
21309062

Sorbonne Université
2024-2025

Contents

1 Introduction

This project focuses on optimizing an application called `MUrB`, which simulates the movement of planets (bodies) over time. This simulation represents a classic problem in Newtonian mechanics known as the `N-body problem`, where the gravitational forces between bodies are computed to determine their positions and velocities. The problem is embarrassingly parallel, as the force calculations for each body are independent, making it ideal for parallelization.

The primary goal of this project is to optimize the CPU version of the MUrB application by implementing multiple parallelization and optimization strategies. Starting with the CPU version, we progressively introduced SIMD for vectorized computations, OpenMP for multi-threading, a GPU-based acceleration and finally an hybrid version using both the CPU and GPU to achieve significant speedups.

The computation consists of determining the gravitational forces between bodies and updating their positions and velocities over discrete time steps. Without adjustments, the gravitational force grows infinitely large as two bodies get very close, leading to numerical instabilities. To address this, a softening factor is introduced, which limits the magnitude of the force and ensures the simulation remains stable and realistic.

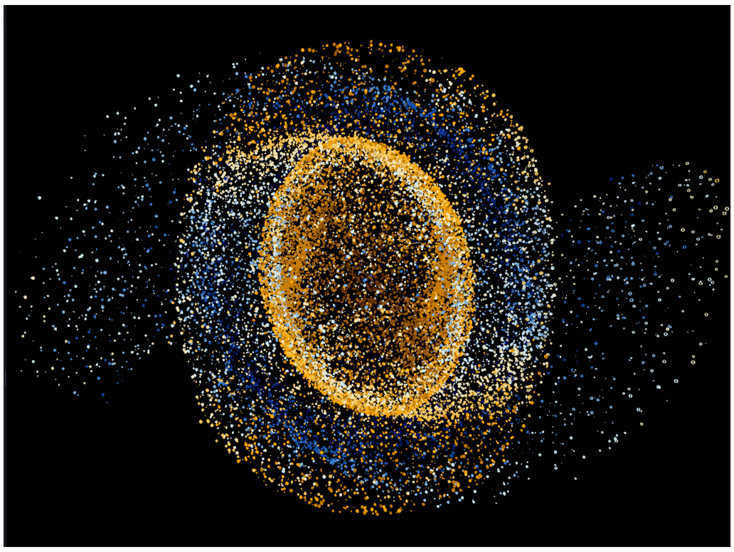

Figure 1: Graphical execution with 30k bodies

2 Description of the Optimizations

2.1 CPU

2.1.1 Triangulation of the calculations (cpu+optim1)

The original computation use a nested loop to calculate interactions between all pairs of bodies. For each body, it computes its interactions with all other bodies.

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < N; j++) {
    computeBodiesInteraction(i, j);
  }
}
```

Listing 1: Original n-bodies loop

By exploiting the symmetry in the bodies interactions, we can reduce the amount of computations and memory loads. We modify the loop to process each pair once, thus using the already loaded j-body, and we reuse the distance computation for both accelerations.

```
for (int i = 0; i < N; i++) {
  for (int j = i + 1; j < N; j++) {
    computeBodiesInteraction(i, j, distance);
    computeBodiesInteraction(j, i, distance);
  }
}
```

Listing 2: Triangulated n-bodies loop

2.1.2 Approximation functions (cpu+optim1_approx)

This optimization improves the performance of the mathematical calculations. The `pow` function from the standard C++ library, used to compute $a^{\frac{3}{2}}$, is inefficient because it requires multiple iterations. We can simplify $a^{\frac{3}{2}}$ as $a * \sqrt{a}$, and further to $\frac{1}{\sqrt{a}^3}$, avoiding the need for `pow`.

```
// compute the acceleration value between i and j bodies
const float rsqrt = 1 / sqrt(rijSquared);
const float rsqrt3 = rsqrt * rsqrt * rsqrt;
const float accelerationFactor = this->G * d[jBody].m * rsqrt3;
```

Listing 3: Sqrt optimization

2.2 SIMD naive, reduction of the j-bodies loop

2.2.1 Reduction of the j-bodies loop (simd+naive)

In our architecture, the SIMD registers are 128 bits wide, allowing us to store 4 single-precision floats (32 bits each) per register. Using these registers, we can reduce the number of iterations for the j-bodies loop by calculating 4 accelerations simultaneously.

To load 4 i-body positions at once, we use a **Structure of Arrays (SoA)** format. This enables us to load 4 consecutive x, y and z positions from their respective arrays. In contrast, using **Array of Structures (AoS)** would result in loading data from the body structure, so x, y, z and vx from the first body instead of $x0$, $x1$, $x2$, $x3$ as described in Figure 2.

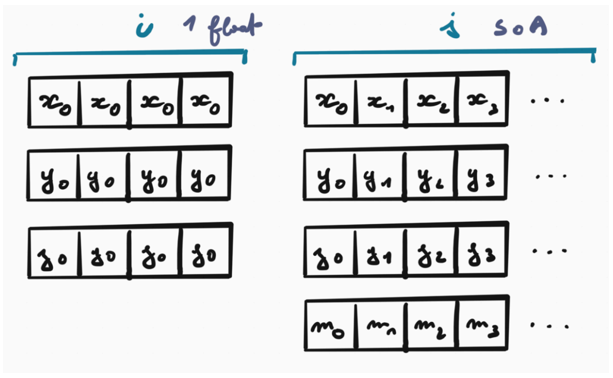

Figure 2: J-bodies SIMD register organisation

Moreover, we apply the following optimizations compared to the naive cpu version, a simplification of the optimized code is described in Listing 4.

Accelerations accumulation

Instead of adding and writing the acceleration for each j-body iteration, we accumulate the values directly in the j-loop and perform a single write operation at the end. This reduces the number of store operations and make better use of the simd register, because we don't have to reduce it to a single element at each iteration.

Deferred multiplication by G

The multiplication by G is deferred until the end of the j-loop, which avoids redundancy. This is valid as this operation is commutative, and is in fact the original description in the mathematical equation from the subject:

$$\vec{a}_i(t) = \frac{\vec{F}_i(t)}{m_i} \approx G \cdot \sum_{j=1}^{n} \frac{m_j \cdot \vec{r}_{ij}}{\left(\|\vec{r}_{ij}\|^2 + \epsilon^2\right)^{\frac{3}{2}}}$$

Horizontal addition

After accumulating the accelerations in the SIMD registers for 4 differents j-body at once, we need to write it for a single i body. To do so, we perform an horizontal addition, which is better because of its logarithmic complexity and implementation, using shuffle and grouped addition instead of lane accesses which are really slow in SIMD.

```
for (int i = 0; i < N; i++) {
  Reg<float> regAcceleration;
  for (int j = 0; j < SIMD_N; j += 4) {
    regAcceleration += computeAcceleration(i, j);
  }
  acceleration[i] += mipp::sum(regAcceleration) * G;
}
```

Listing 4: J-bodies SIMD loop

2.2.2 Usage of reverse sqrt (simd+optim1)

Additionaly, we use the **Fast Inverse Square Root Algorithm**, originally from the Quake team and now implemented efficiently in hardware, which approximate $\frac{1}{\sqrt{a}}$ much faster. By replacing the `1 / sqrt` operation in Listing 3 with `rsqrt`, we get way better performance.

Precision issue with ARM SIMD

On ARM architectures, the SIMD NEON intrisic `vrsteq_f32` provides an estimate of the inverse square root. However, its precision is insufficient to pass our tests. To fix this issue, we add an additional step using `vrsqtsq_f32`, which applies the **Newton-Raphson** method step by step to produce a better result based on the original approximation.

Based on our tests, we observe that one step achieves sufficient accuracy, with an approximate 20% performance trade-off. The final implementation is described in Listing 5.

```
template<>
inline reg rsqrt_prec<float>(const reg v1) {
  float32x4_t approx = vrsqrteq_f32(v1);
  return vrsqrtsq_f32(v1 * approx, approx) * approx;
}
```

Listing 5: Rsqrt optimization with additional step

This implementation is available in a pull request for MIPP on the `master-rsqrt-prec` branch: MIPP PR 59. Thus, the MIPP submodule commit reference in `murb` must be updated to include support for `mipp::rsqrt_prec`.

2.2.3 Reduction of the i-loop (cpu+optim2)

The main idea behind this optimization is to reduce cache misses in the j-loop. When using a **SoA** layout, we are inducing cache misses due to frequent switching between arrays. To address this, we introduce the `packedAoS_t` structure.

This structure, described in Listing 6, is similar to **AoS** but contains only the fields needed by the algorithm. By reducing the unused data, `packedAoS_t` is 2x smaller than the original structure, thus minimizing unnecessary memory loads and cache line thrashing.

```
struct packedAoS_t {
  float qx; /* position x. */
  float qy; /* position y. */
  float qz; /* position z. */
  float m; /* mass. */
};
```

Listing 6: Packed data structure

Shuffle of the j-body components

Using the `packedAoS_t` structure in the j-loop, which is 4-floats sized, we can load an entire j-body in a SIMD register at once in an **AoS** way. Afterward, instead of fetching each component of the lane one by one (which is inefficient), we use a **shuffle operation**. By creating 4 different `mipp::cmask`, we can create 4 seperates registers for the x, y, z and m component of the j-body.

Simultaneously, we load 4 i-bodies at once in a **SoA** way, component by component. We then calculate the interaction between 4 i-bodies and 1 j-body, as illustrated in Figure 3.

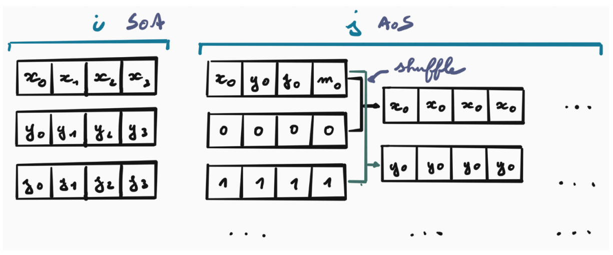

Figure 3: I-bodies SIMD register organisation

Further optimizations

Building on earlier SIMD optimizations, we accumulate the accelerations instead of writing them at each j-iteration. Moreover, processing 4 i-bodies at once mean that we can eliminate the horizontal reductions, as the 4 accelerations can be stored in one step. This is shown in Figure 3.

```
for (int i = 0; i < SIMD_N; i += 4) {
  Reg<float> regAcceleration;
  for (int j = 0; j < N; j++) {
    regAcceleration += computeAcceleration(i, j);
  }
  regAcceleration.store(&acceleration[i]);
}
```

Listing 7: I-bodies SIMD loop

2.3 OpenMP version

We implemented OpenMP on top of our best SIMD version (SIMD+optim1) to implement multi-threading.

We chose to use a dynamic scheduler over a static one because it provides better workload balancing by dynamically distributing tasks in real time during loop execution.

We also had to determine how many iterations each thread should handle. After experimentation, we found that using $\frac{\text{Nb\_bodies}}{60}$ iterations per thread delivered the best performance.

Additionally, we set all the parameters to `private` instead of `shared` because none of them required to be shared accross the threads. This decision contributed to improved performance with a 17x speedup. An example of the implementation is shown below:

```
#pragma omp parallel for schedule(dynamic, N / 60) firstprivate(d, reg_qx, reg_qy, reg_qz, ...)
```

After applying all these optimizations, we wanted to determine the optimal number of threads to use. To achieve this, we conducted an experiment to evaluate performance with different thread counts, as illustrated in the results obtained with Figure 4.

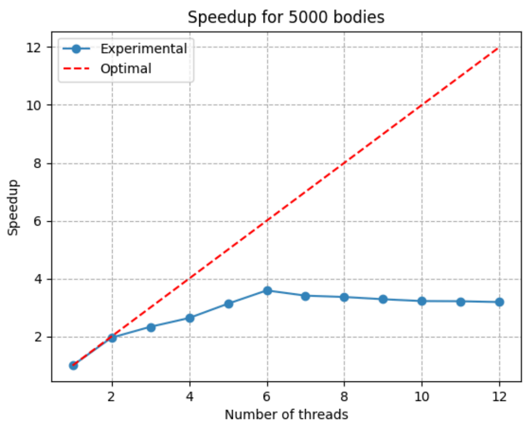

Figure 4: OMP optimization for 5K bodies per number of threads

Cores	FPS
Denver 1-core	17.3
Cortex 1-core	7.8
Denver cluster (2 cores)	34.6
Cortex cluster (4 cores)	30.4
All cores	58.1

Table 1: OMP performance per cluster and per core for 5000 bodies

Like we can see in Figure 4, the best number of threads is 6. This result it expected since we have 6 physical cores. However, we do not observe a 6x speedup with 6 threads. This can be explain by the heterogeneity on the system. The Denver and Cortex don't have the same performance, likely because the Denver cores have larger data cache sizes. An overview of the performance per cluster and per core is detailed in Table 1.

Moreover, the parallelization factor is dependant on the sequential part of the program, as dictated by Amdahl's law. While dynamic scheduling is faster than static scheduling, it introduces additional overhead because indices are assigned in real-time.

2.4 GPU version

2.4.1 Naive version (cuda+naive/ocl+naive)

The naive GPU implementation adapts the algorithm to the GPU's parallel architecture by assigning one thread to each body. Each thread then loads all other bodies to compute interactions. This removes the need for the i-loop, as the GPU can dispatch thousands / millions of threads concurrently. The algorithm is described in Listing 9.

```
// The calculation below find the id of the thread, based on the block id.
// This is unique for the 30k threads.
iBodyData = loadIBodyData(blockDim.x * blockIdx.x + threadIdx.x);
for (int jBody = 0; jBody < nBodies; j++) {
  computeBodyInteractions(jBody, iBodyData);
}
```

Listing 9: Naive GPU algorithm

Block size optimization

To fully use the GPU, we need to find the best block size for the naive version. On the jetson TX2, there are 2 Streaming Multiprocessors (SMs), each capable of managing up to 1024 threads. Threads are grouped into warps, with each warp consisting of 32 threads. This is why the block size dimension must be a multiple of 32 and can range from 32 to 1024.

We tested multiple block sizes to maximize occupancy and performance. The results are summarized in Table 2, the command used is described in Listing 10. From the result, smaller block size achieves better occupancy and generally better performance because they allow more flexibility for the GPU scheduler. However, it's surprising that a block size of 32 result in only 49% occupancy.

```
/usr/local/cuda-10.0/bin/nvprof --print-gpu-trace --metrics achieved_occupancy ./bin/murb -n 30000
-i 200 --im cuda+naive --nv
```

Listing 10: Command using `nvprof` to get the occupancy percentage

Block size	FPS	Occupancy percent
32	9.67	49%
64	10.57	96%
128	10.55	96%
256	10.56	95%
512	10.10	93%
1024	10.21	90%

Table 2: GPU occupancy and FPS per block size with 30k bodies

2.4.2 Usage of shared memory (cuda+optim1)

The naive version is memory-bound due to slow GPU fetch operations from the global memory: each thread needs to fetch all the bodies. To address this, we exploit the faster but smaller shared memory available on each SM. However, our architecture's shared memory is limited to 48 KB, meaning that we can store at most 3000 packed bodies (using `packedAoS_t`, with each body consisting of 4 single-precision floats, calculated as $\frac{48000}{4 \cdot 4} = 3000$).

Since our benchmark simulation consist of 30k bodies, all the bodies cannot fit in shared memory. The solution is to group the interaction calculations for the j-bodies into multiple passes. During each pass, each thread loads one body into the shared memory. Afterward, all threads synchronize, allowing every thread in the block to access the shared memory and compute interactions with the current set of j-bodies for this pass. The memory organization is further described in Figure 5.

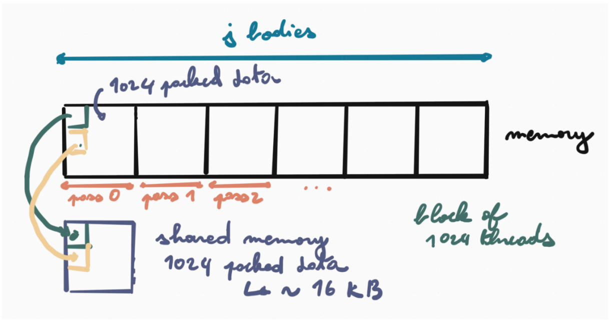

Figure 5: Shared memory organization

Additionally, we optimized shared memory usage by doubling the data loaded in each pass, increasing from 1024 to 2048 bodies. Each thread loads two bodies in a coalesced manner to maximize architectural efficiency, this is described in Listing 11. However, further increasing shared memory usage reduces performance. We are not entirely sure why, but it could be due to more bank conflicts in shared memory or other memory-related effects we don't fully understand.

```
for (int pass = 0; pass < nbPass; pass++) {
  // threadIdx.x is the id of the thread in the block, from 0 to 1024.
  // shBodies represents the packedAoS_t bodies array in the shared memory.
  // +1024 instead of blockIdx * 2 + (0, 1) for better coalescing.
  shBodies[threadIdx.x] = inBodies[startPassIdx + threadIdx.x];
  shBodies[threadIdx.x + 1024] = inBodies[startPassIdx + threadIdx.x + 1024];
  __syncthreads();
  for (int jBody = startPassIdx; jBody < endPassIdx; jBody++) {
    computeBodyInteractions(jBody, iBodyData, shBodies[jBody]);
  }
}
```

Listing 11: GPU with shared memory

In this version, we use a block size of 1024, as we found out that it gives the best performance compared to smaller block sizes. We believe this is because memory latency is not directly tied to occupancy, and using larger passes with more threads that load more shared memory leads to better performance due to the reduced memory latency.

2.4.3 Computation of two i-bodies per thread (cuda+optim2)

This optimization computes interactions for two i-bodies in one thread. By reusing data loaded in the j-loop, we reduce memory latency and improve compute usage. However, this approach increases the register pressure, necessitating a reduction in block size to 768 threads. This also modifies the pass logic to treat the jbodies loop by pass of size of 1536, meaning that our shared memory is now 1536 bodies elements, as shown in Figure 6.

Our tests showed that processing more than two i-bodies per thread is counterproductive, as the increase of registers requires reduction of the block size, leading to lower occupancy and performance.

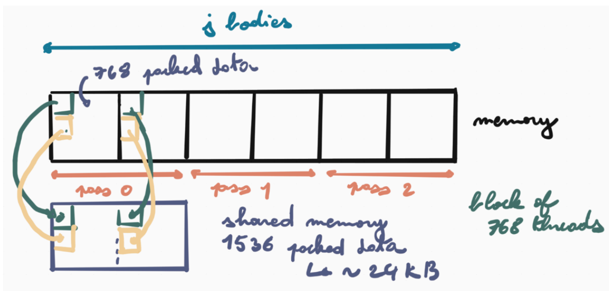

Figure 6: Shared memory with 768 threads

2.4.4 Usage of SoA instead of AoS (cuda+optim3)

To achieve better memory coalescing on the GPU, we replaced the **AoS** layout with **SoA**. On GPU architecture, using the **AoS** layout is inneficient because each instruction is run by multiple threads, meaning that when we load the x component, this requires loading unnecessary data (y, z and m), thus wasting memory bandwidth.

With this optimization, we replaced `packedAoS_t` with a **SoA** layout and separated the shared memory into four arrays for each component (x, y, z and m).

2.5 Hybrid version

2.5.1 OpenCL with MIPP version (hetero)

We implemented a heterogeneous version using OpenCL instead of CUDA because NEON intrinsics are not compatible with NVCC.

We observe that running the kernels on the GPU is asynchronous, and while the GPU is executing, the CPU cores remain mostly idle, waiting for the `clFinish(command_queue)` call.

In this optimization, we split the workload between the GPU and the CPU. Instead of waiting for the GPU to complete its task, we offload part of the work to the CPU, allowing both to work in parallel.

The workload is divided based on a specific ratio. The GPU handle the first portion of the bodies, with less threads assigned in the grid. The CPU process the remaining bodies, starting from an offset in the SIMD loop, as described in Listing 12.

```
// launch the GPU kernel.
computeBodiesAccelerationGPU();
// Compute the remaining bodies on the CPU using SIMD, starting from the given index.
computeBodiesAccelerationCPU(startBodyIdx);
// Wait for the GPU to finish its execution and memcpy the results in the CPU.
clFinish(command_queue);
clEnqueueReadBuffer(...)
```

Listing 12: Hybrid computation with CPU and GPU

We then tested different workload ratios between the GPU and CPU, as detailed in Table 3. Allocating 10% of the workload to the CPU yields the best result, resulting in a 15.8% FPS increase compared to the GPU-only configuration. This corresponds to a speedup of 1.16x. However, allocating 20% of the workload led to a 12.4% FPS decrease. This suggest than giving too much work to the CPU has a negative impact on the performance, likely due to the increased CPU processing time compared to the GPU.

The performance could be further improved by running the GPU queue initialization and memory copy operations on a separate thread. This would allows true concurent GPU and CPU operations.

CPU workload (%)	**FPS**	**FPS change (%)**
0%	10.18	+0%
1%	10.30	+1.2%
5%	11.30	+11.0%
10%	11.81	+15.8%
15%	11.22	+10.2%
20%	8.92	−12.4%

Table 3: Percentage of workload in the CPU for the CPU / GPU hybrid version with 30k bodies

3 Computer Architecture

The experiments were performed on an **NVIDIA Jetson TX2**, a platform combining CPU and GPU components for heterogeneous computing.

3.1 CPU Architecture

The Jetson TX2 CPU has six cores organized in two clusters:

- **Denver 2**: A dual-core CPU designed for higher single-thread performance.
  - 7-wide super-scalar, 2.5 GHz.
  - 64 KB L1 data cache, 128 KB L1 instruction cache, 2 MB unified L2 cache.
- **ARM Cortex-A57 MPCore**: A quad-core CPU optimized for multi-threaded applications and lighter loads.
  - 8-wide super-scalar, 2.5 GHz.
  - 32KB L1 data cache, 48KB L1 instruction cache, 2MB unified L2 cache.

The multi-clusters architecture is illustrated in Figure 7.

The CPU uses the ARMv8 architecture with SIMD operations via NEON intrisics and 128-bit registers for parallel data processing. Based on each clusters' cores frequency, super-scalar capabilities, and the fact that one SIMD register can hold 4 single-precision floats, the maximum compute performance is estimated at 375 GFlops.

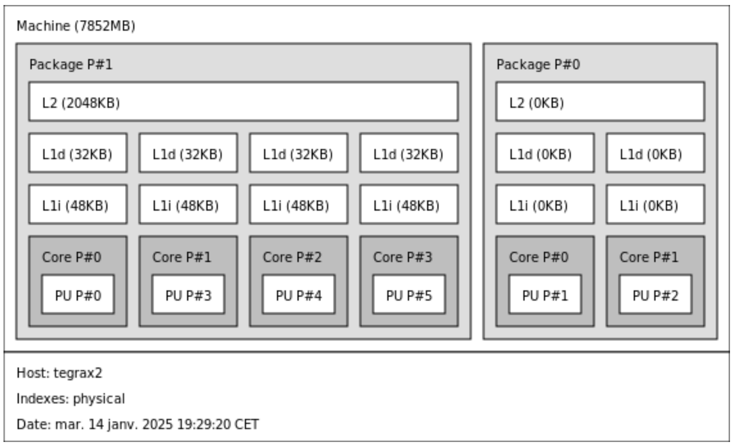

Figure 7: CPU architecure with hwloc

3.2 GPU Architecture

The GPU within the **Jetson TX2** is based on the NVIDIA Pascal architecture as described in Figure 8, which features:

- **256 CUDA cores**: 665.6 GFlops for single-precision float.
- **2 Streaming Multiprocessors (SMs)**: Each SM can manage up to 1024 threads simultaneously, with 128 cores.
- **48 KB shared memory per SM**: Reduce global memory access time, used for faster load in redundant loads scenario.
- **32 K registers per block**: The more registers we use, the less threads we can run in a block.

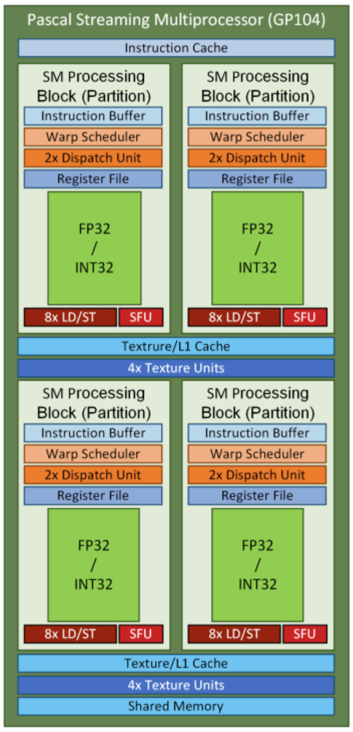

Figure 8: GPU Pascal architecture

3.3 Shared Memory Architecture

The **Jetson TX2** features 8GB of LPDDR4 shared memory with a bandwidth of 58 GB/s. Unlike discrete memory systems, its unified memory architecture enables efficient data sharing between the CPU and GPU, eliminating the need for memory copies. A comparison of both memory models is shown in Figure 9.

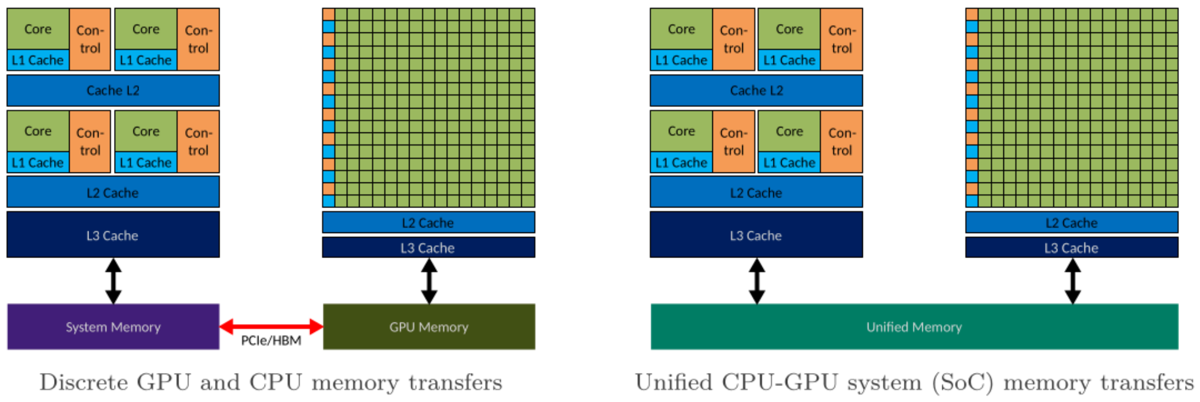

Figure 9: GPU Pascal architecture

4 Experimentation Results

We obtained our data to plot the graph by running all the simulations 10 times and taking the average and standard deviation.

4.1 CPU version

We performed all the CPU experimentation on 1000 bodies.

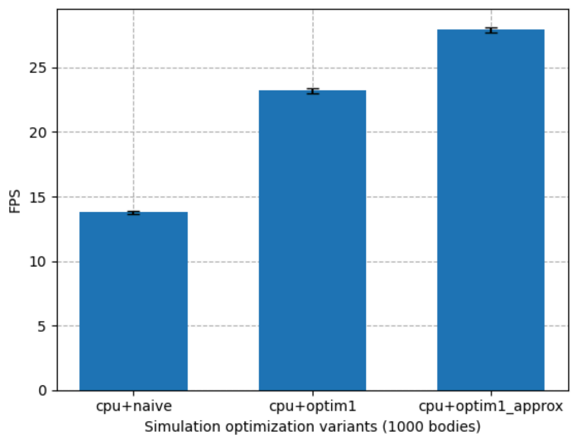

Figure 10: CPU optimization with 1K bodies per FPS

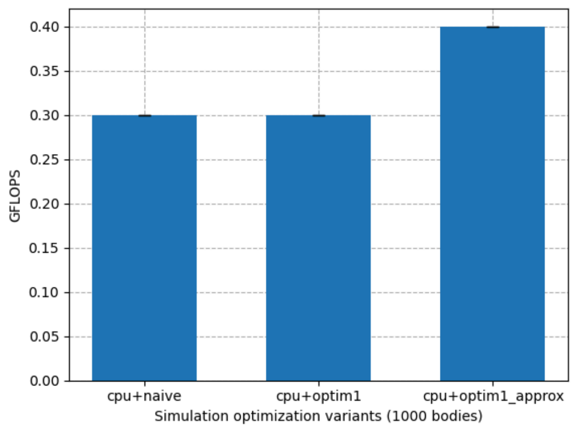

Figure 11: CPU optimization with 1K bodies per GFlops

As shown in Figure 10, the triangulated version (`cpu+optim1`) achieves a 1.7x speedup. This improvement can be explained by the reduction in the number of iterations and the reuse of computations in the j-loop. Additionally, as shown in Figure 11, the GFlops remain constant, indicating that we reduced memory latency by making better use of already-loaded elements and computing two bodies at once in the j-loop.

In contrast, replacing `pow` with `sqrt` result in a smaller peformance gain, achieving a 1.2x speedup.

4.2 SIMD version

We implemented our SIMD version using MIPP instead of using the NEON intrisics, as MIPP offers compatibility with a broader range of architectures. All SIMD experiments were done with 10,000 bodies.

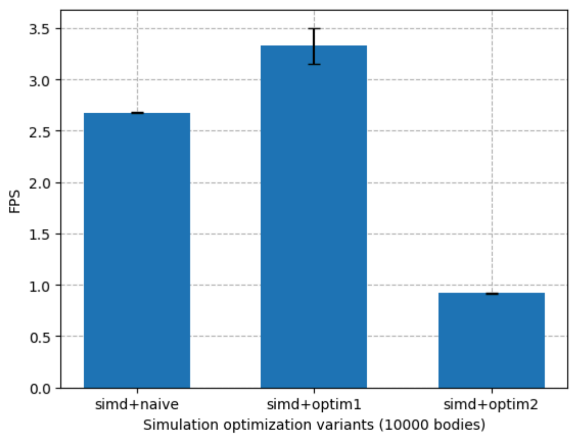

Figure 12: SIMD optimization with 10K bodies per FPS

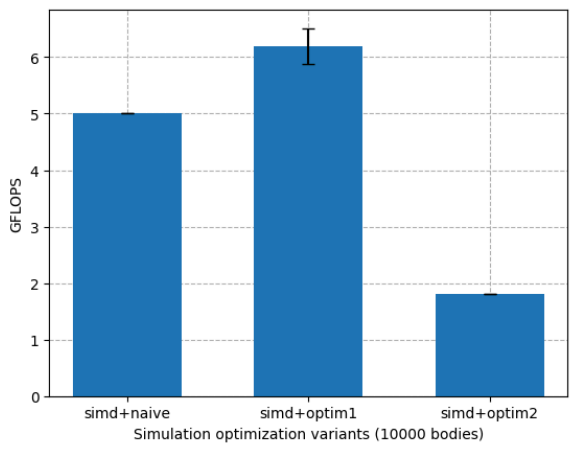

Figure 13: SIMD optimization with 10K bodies per GFlops

We observed a 1.4x speedup in the best `simd+optim1` version compared to the naive one. This results is better than the CPU version, thanks to the use of the native `rsqrt` estimate instruction. However, the second optimization did not achieve the expected results, performing approximately 3 times worse. This might be attributed to a suboptimal implementation on our part. Finally, as shown in Figure 13, the performance is still far from the theoretical maximum of 375 GFlops, with the peak in simd+optim1 reaching only 6 GFlops.

4.3 GPU version

All GPU experiments were done with 30,000 bodies.

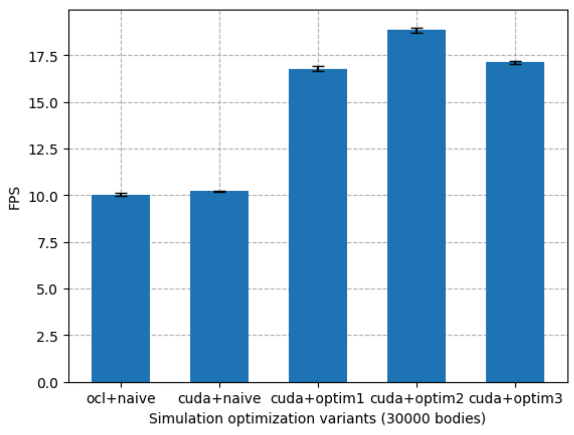

Figure 14: GPU optimization with 30K bodies per FPS

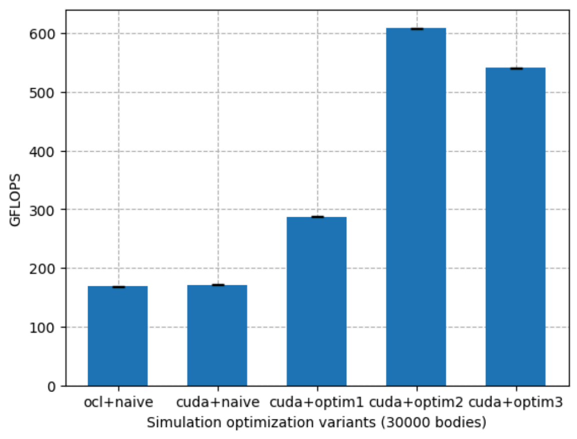

Figure 15: GPU optimization with 30K bodies per GFlops

We implemented the naive version of the simulation using both OpenCL and CUDA and tested their performance. For test cases with 1,000 - 5,000 bodies, we observed a noticeable performance difference, with OpenCL being approximately 20% slower. The difference is mainly due to initialization and kernel launch overhead. However, for a larger test case (30,000 bodies), as shown in Figure 14, both implementations performed similarly, with computation dominating the total runtime. As a result, performance differences at smaller scales are less critical. OpenCL's main advantage is its portability across various architectures, while CUDA is limited to NVIDIA GPUs.

Adding shared memory (`cuda+optim1`) resulted in a 1.64x speedup despite a 1.68 Gflops increase (shown in Figure 15). This is because shared memory access is faster than global access.

In `cuda+optim2`, each thread compute 2 i-bodies, leading to a 1.12x speedup and 2.1x Gflops increase over the `cuda+optim1` version. This is due to the fact that this optimization increased the computation per thread, reaching the optimal GPU compute capabilities.

Finally, we switched from an Array of Structures (AOS) to a Structure of Arrays (SOA) layout to improve memory coalescing. However, for 30,000 bodies, this approach was slower than `cuda+optim2`, reaching 0.9x the original FPS and GFlops. This might come from the shared memory architecture between the GPU and CPU.

4.4 Fastest version

Finally, we compared the fastest versions of each group of optimizations on 10,000 bodies.

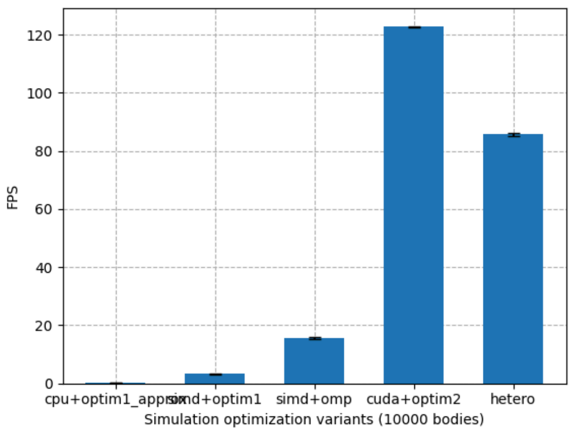

Figure 16: Fastest optimization with 10K bodies per FPS

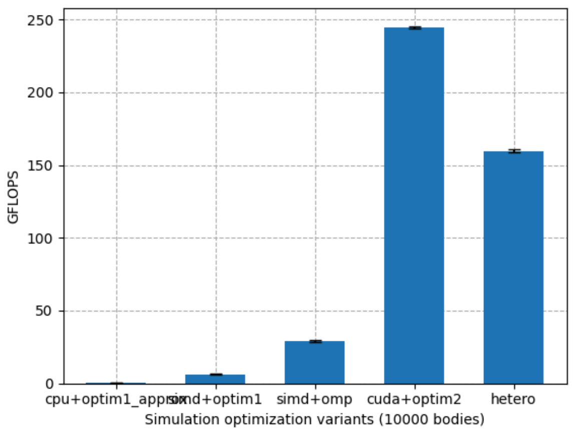

Figure 17: Fastest optimization with 10K bodies per GFlops

As shown in Figure 16, the best SIMD version achieved a 12x speedup over the best cpu version. Additionally, the SIMD + OpenMP implementation, using all cores, achieved a 4.7x speedup. The best CUDA version outperformed all others, achieving a 7.9x speedup over the SIMD + OpenMP version and a 441x speedup over the best cpu version. Notably, the heteroneous version, though promising, was implemented using a naive OpenCL version without shared memory optimization, which limited its performance compared to the optimized CUDA version.

5 Improvements

From our benchmark, there's room to improve the SIMD version, as the current performance (6 GFlops, an approximation) is far below the architecture's theoretical peak of 375 GFlops. Better usage of NEON intrisics and advanced register operations, such as multiplying a single lane element at a given index with an entire SIMD register, could improve performance. Additionally, unrolling the i-body loop and reusing the j-body loop across multiple i-bodies could better leverage the cores' out-of-order capabilities and hide memory latency.

For the GPU version, addressing shared memory bank conflicts could resolve the performance drop observed when increasing shared memory usage, likely caused by our implementation. Another improvement would be to avoid unnecessary data copies between CPU and GPU, either by modifying the data structures in common or maintaining a GPU-resident duplicate buffer of the bodies which would be updated on the GPU side.

For the heterogeneous version, future improvements could involve implementing optimizations from our CUDA version into the OpenCL implementation, allowing us to take full advantage of both CPU and GPU resources.

Finally, implementing a more efficient algorithm, such as Barnes-Hut could lead to massive performance gains by reducing the algorithmic complexity from $O(N^2)$ to $O(N \cdot \log_2(N))$. This would allow the CPU to construct the the acceleration structure send it to the GPU, making better use of both architectures.

6 Conclusion

The goal of this project was to optimize the n-body simulation for the Jetson TX2 architecture. Our benchmarks and optimizations validated the fact that the problem is embarrassingly parallel, as each body are independent and most operations are commutative. This allowed us to scale performance with more cores, with the GPU benefiting the most due to its parallel processing capabalities. Memory bandwidth was a primary bottleneck, driving multiple optimizations like the use of shared memory on the GPU.

We found that fully utilizing the SIMD capabilities of the architecture led to significant performance gain, with library like MIPP enabling portability accross architectures. Moreover, using all the CPU cores resulted in a 4.7x speedup with six cores, showing minimal sequential limitations and aligning with Amdahl's law.

The GPU outperformed the CPU, achieving nearly a 10x speedup over the best SIMD multi-cores implementation and a 441x speedup over the best single-threaded cpu optimization (without SIMD). Moreover, heterogeneous computing, using both the GPU and CPU simultaneously, can lead to better performance, with a 15% performance increase compared to a single GPU version.

We conclude that GPUs are the best choice for this type of problem. In case where GPUs are unavailable, careful use of SIMD intrisics and multi-core parallelism, such as with OpenMP, can still reach good performance. Additionally, leveraging both CPU and GPU through heterogeneous computing can further enhance performance.